Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 10: Tuesday 4/11 @ 11:59 pm
  - Homework 11: this Friday @ 11:59 pm
  - Assignment 4: this Friday @ 11:59 pm
    - Support video and slides on Canvas + Solutions for Labs 8 and 9
  - Midterm Question Reattempts: Monday 4/17 @ 11:59 pm
    - up to **7 points** back
    - Please use GradeScope's Regrade Requests for each question **individually**

Previous Lecture …

Weighted Shortest Paths problem

- Dijkstra's single-source shortest paths algorithm
  - Real-world optimizations
- Bellman-Ford's shortest paths algorithm
  - correct with negative edge weights
  - negative cycles

This Lecture …

Dynamic Programming

- A recipe
- Examples:
  - Computing the n^{th} Fibonacci Number
  - Unbounded Knapsack

Let’s change focus into a different method of problem solving

We will get back to graphs in the last week!

Consider computing the nth Fibonacci number

+

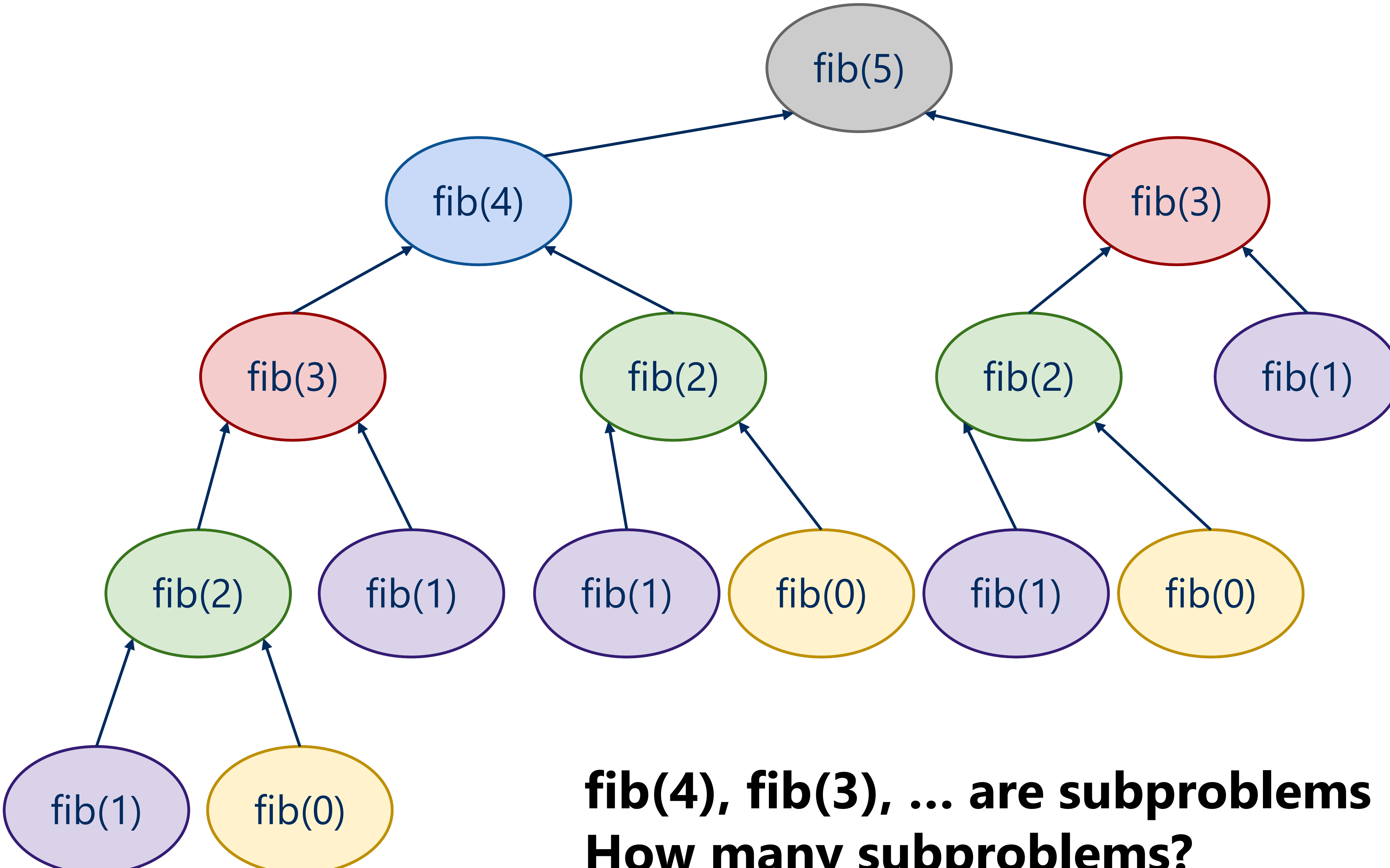
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, - -

+

```
int fib(n) {
     if (n == 0) { return 0 };
     else if (n == 1) { return 1 };
     else {
          return fib(n - 1) + fib(n - 2);
     }
}
```

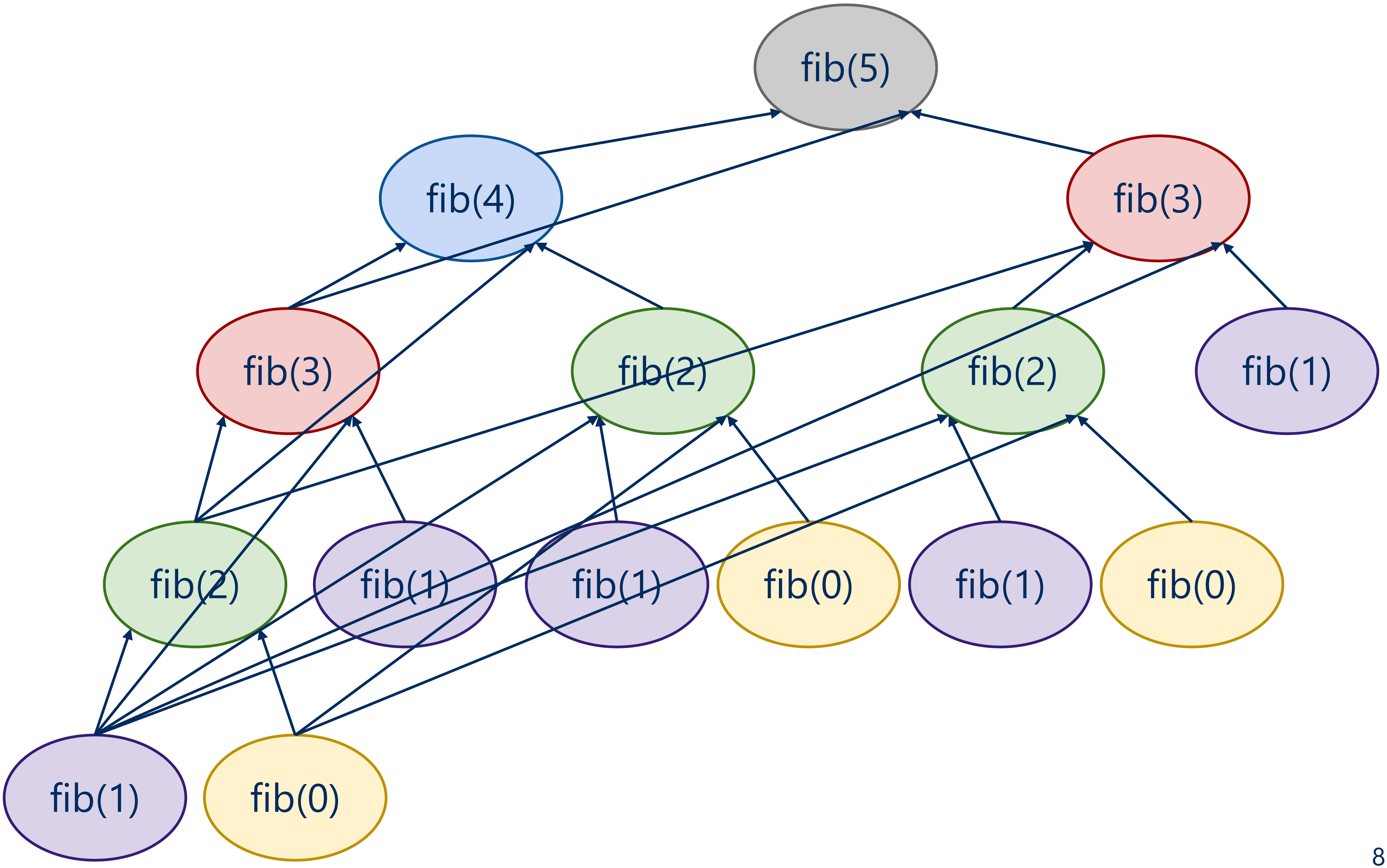
- What is the running time?
- What does the call tree for n = 5 look like?

fib(5)



fib(4), fib(3), … are subproblems
How many subproblems?

How do we improve?



Memoization: save solutions for solved subproblems

```
int[] F = new int[n+1];
F[0] = 0;
F[1] = 1;
for(int i = 2; i <= n; i++) { F[i] = -1 };

int fib_mem(n) {
   if (F[n] == -1) {
        F[n] = fib_mem(n-1) + fib_mem(n-2);
   }
   return F[n];
}
```

- Each subproblem solved once!
- What is the running time?

Note that we can also do this bottom-up!

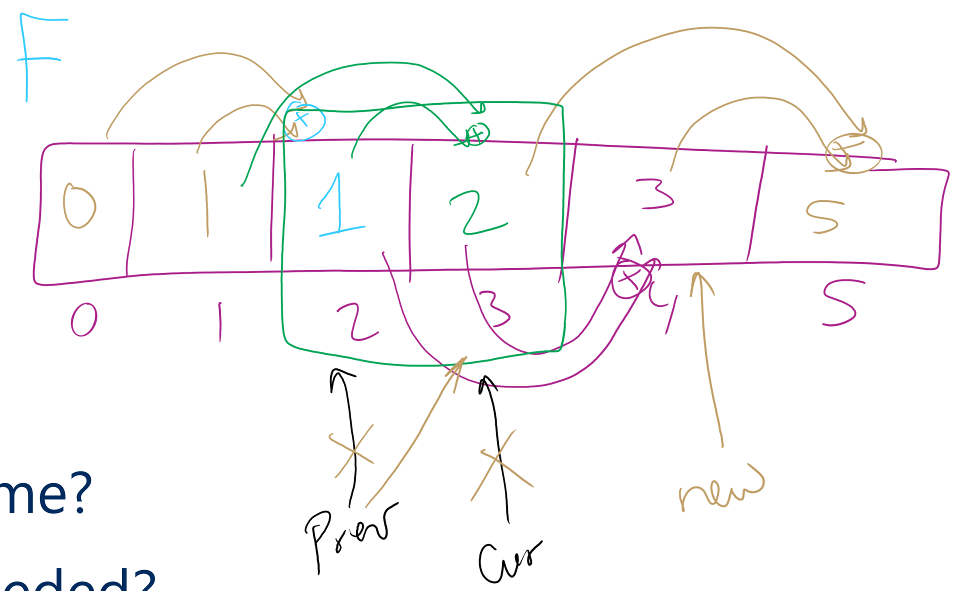
```
int[] F = new int[n+1];
F[0] = 0;
F[1] = 1;

int bottomup_fib(n) {
      for(int i = 2; i <= n; i++) {
            F[i] = F[i-1] + F[i-2];
      }
      return F[n];
}
```

- Each subproblem solved once!
- What is the running time?
- How much space is needed?

Can we improve this bottom-up approach?

```
int improve_bottomup_fib(n) {
      if (n == 0) return 0;
      if (n == 1) return 1;
      int prev = 0; int cur = 1;
      for (int i = 0; i < n; i++){
            int new = prev + cur;
            prev = cur;
            cur = new;
      }
      return cur;
}
```

- What is the running time?
- How much space is needed?



Recap …

- Dynamic Programming
  - avoid solving the same subproblem twice
  - iterative:
    - start with smaller subproblems then larger subproblems, …
  - sometimes possible to optimize space needed

Recap …

- Fibonacci
  - started with inefficient recursive solution
    - solves same subproblems multiple times
  - memoization solution:
    - efficient: solves each subproblem once
    - still recursive
  - dynamic programming:
    - efficient: solves each subproblem once
    - iterative
    - allows for space optimization

Dynamic Programming: a recipe

- What is the **first decision** to make to solve the problem?
  - add fib(n-1) + fib(n-2)
- What **subproblem**(s) emerge out of the that first decision?
  - fib(n-1) and fib(n-2)
- Must **wait** for subproblem solutions to make the first decision?
  - Yes
- start with a recursive solution
- if inefficient, do you have **overlapping** subproblems?
- identify the **unique** subproblems
- Allocate an **array** to hold their solutions
- solve them from **bottom-up** smaller to larger
- Optimize space if possible

Example 2: The unbounded knapsack problem

- a **knapsack** that can hold a **weight limit** L
- a set of n **item types**
  - each has a weight (w_i) and value (v_i)
  - **unbounded** supply of all types
- what is the **maximum value** we can fit in the knapsack?

10 lb. capacity

weight:	6	3	4	2
value:	30	14	16	9

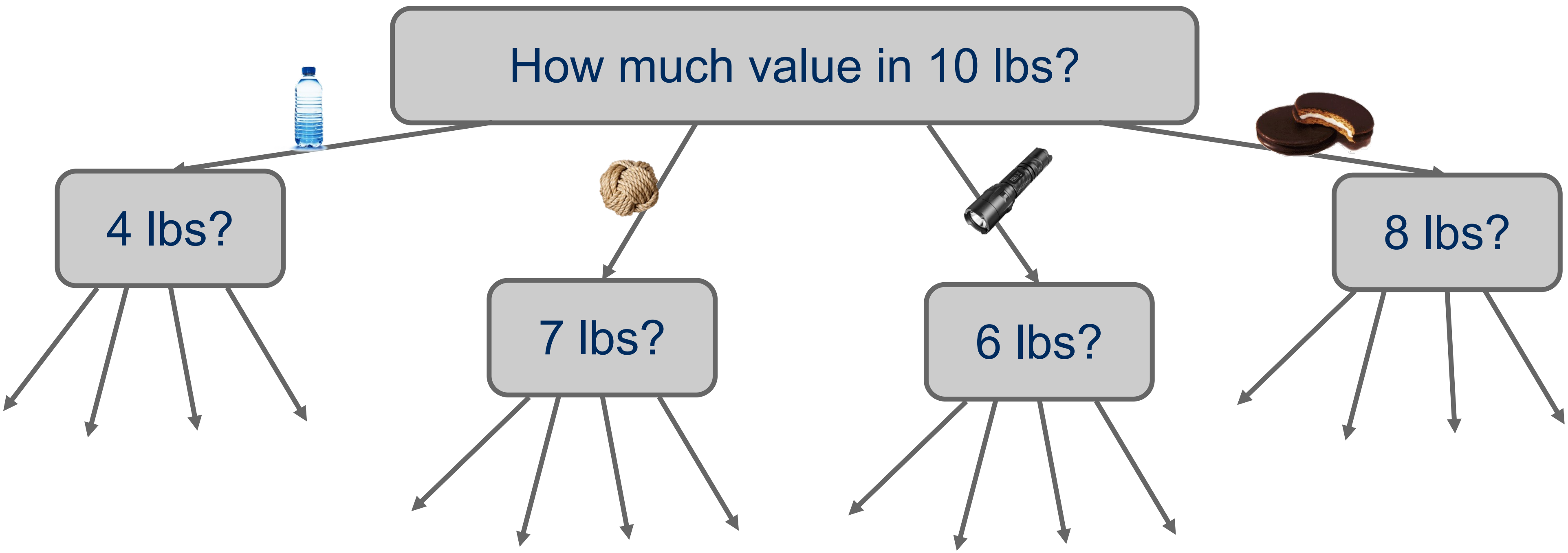
Dynamic Programming: a recipe

- What is the **first decision** to make to solve the problem?
- What **subproblem**(s) emerge out of the that first decision?

Decisions

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Dynamic Programming: a recipe

- What is the **first decision** to make to solve the problem?
  - first item to put in the knapsack
- What **subproblem**(s) emerge out of the that first decision?
  - a knapsack with remaining capacity and all items available
- Must **wait** for subproblem solutions to make the first decision?
  - Yes?

Greedy algorithms

- At each step, the algorithm makes a choice that seems to be best **at the moment**
- **Doesn't wait for subproblem solutions**
- Have we seen greedy algorithms already this term?
    - Yes!
    - Building Huffman tries
    - Prim's MST algorithm

A greedy algorithm for Unbounded Knapsack

- Add as many copies of **highest value per pound** item as possible:
  - Water: 30/6 = 5
  - Rope: 14/3 = 4.66
  - Flashlight: 16/4 = 4
  - Moon pie: 9/2 = 4.5
- Highest value per pound item? Water
  - Can fit 1 with 4 space left over
- Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over
- No room for anything else
- Total value in the 10 lb. knapsack?
  - 44
  - Is that optimal?

10 lb.
capacity

weight:	6	3	4	2
value:	30	14	16	9

Greedy algorithm doesn’t work for this problem

No optimal solution includes the locally-optimal choices made by the greedy algorithm

Dynamic Programming: a recipe

- Must **wait** for subproblem solutions to make the first decision?
  - Yes!
- start with a recursive solution

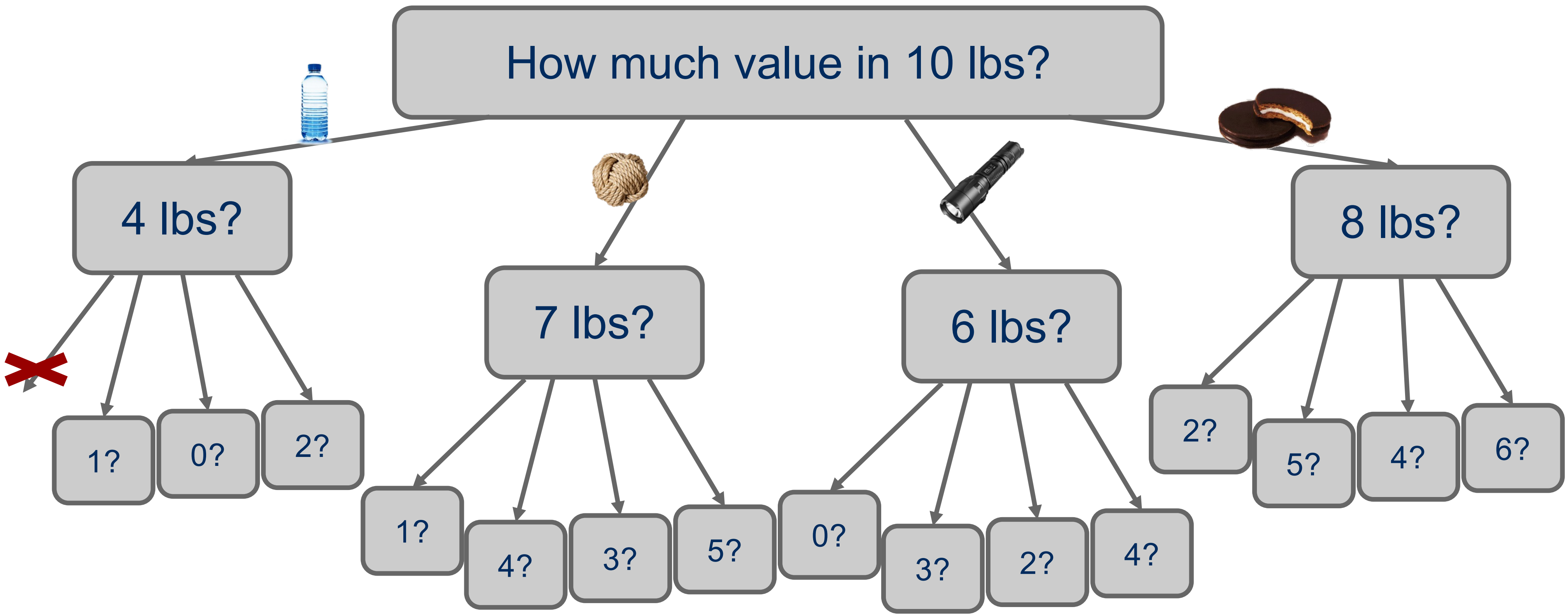
Recursive solution

```
int knapSack(int[] wt, int[] val, int L) {
    if (L == 0) { return 0 };
    int maxValue = 0;
    for(int i=0; i < n; i++){
        if (wt[i] <= L) {
            value = val[i] +
                    knapSack(wt, val, L-wt[i]);
            if (value > maxValue) maxValue = value;
        }
    }
    return maxValue;
}
```

Decisions

weight:	6	3	4	2
value:	30	14	16	9

10 lb.
capacity



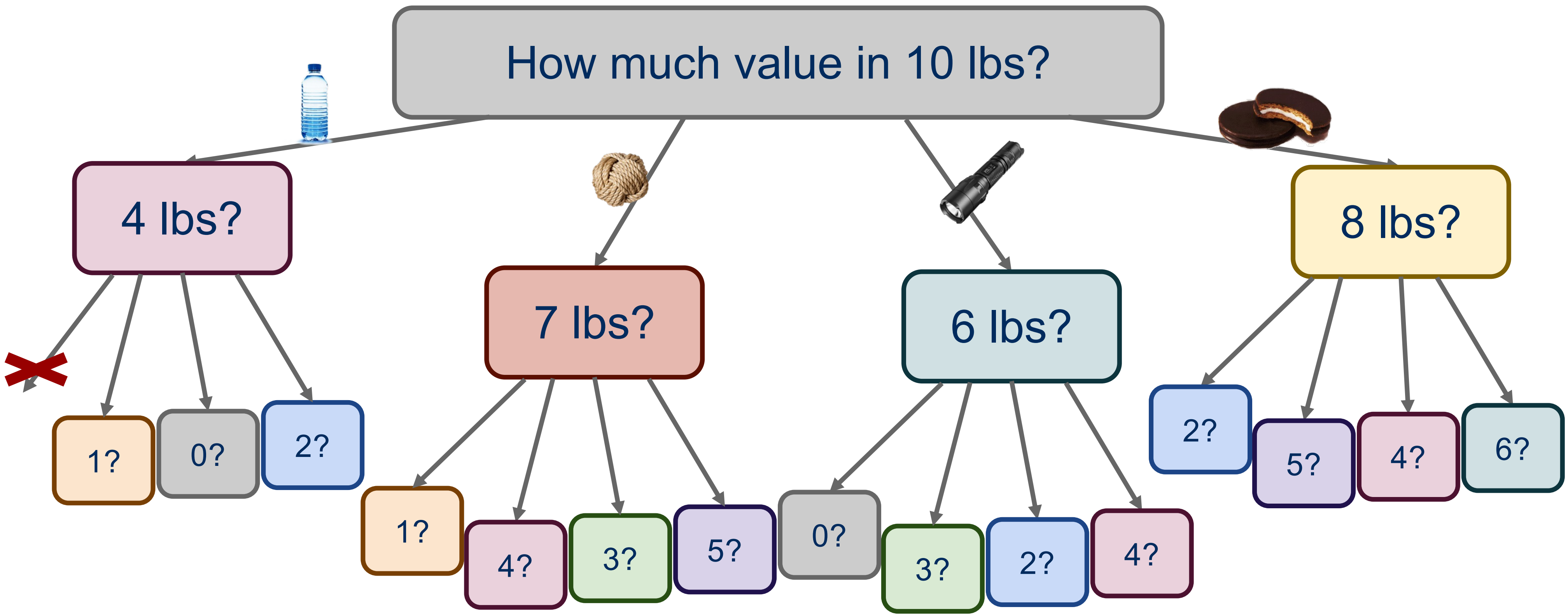
Dynamic Programming: a recipe

- Must **wait** for subproblem solutions to make the first decision?
  - Yes!
- start with a recursive solution
- if inefficient, do you have **overlapping** subproblems?

Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



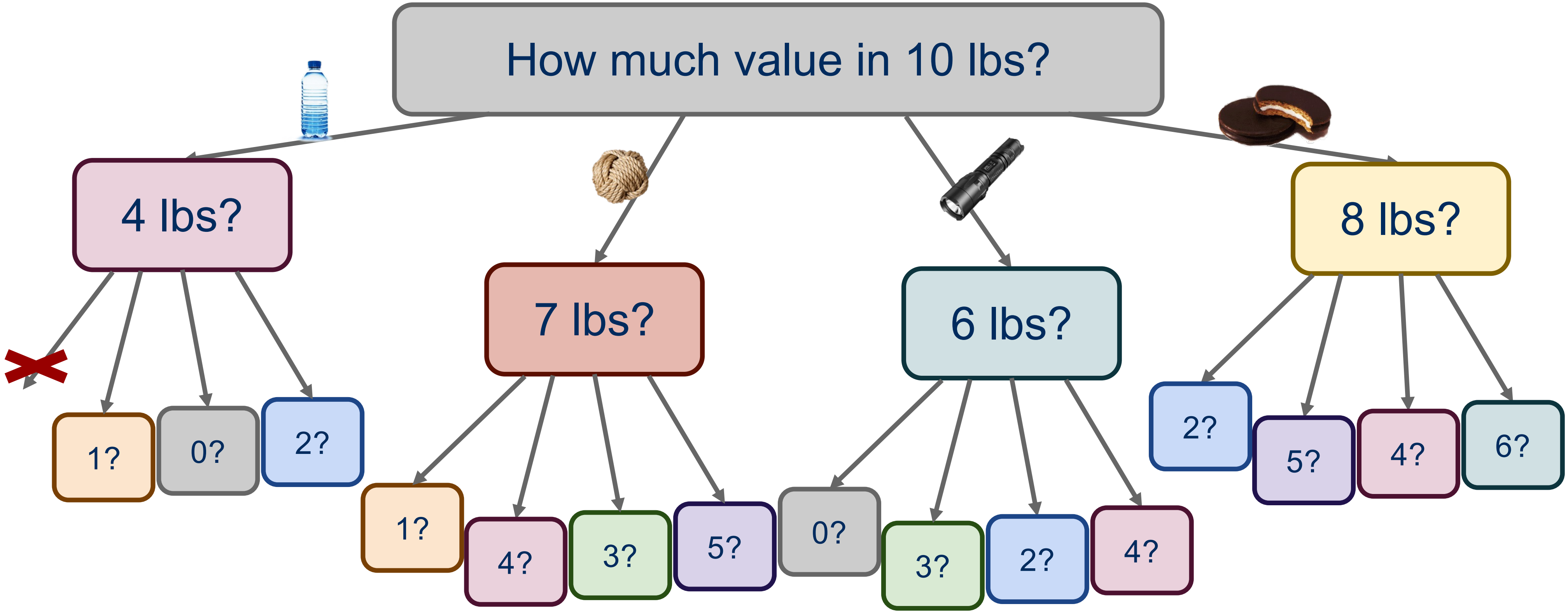
Dynamic Programming: a recipe

- Must **wait** for subproblem solutions to make the first decision?
  - Yes!
- start with a recursive solution
- if inefficient, do you have **overlapping** subproblems?
- identify the **unique** subproblems
- Allocate an **array** to hold their solutions

Ubnique subproblems?

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Dynamic Programming: a recipe

- Must **wait** for subproblem solutions to make the first decision?
  - Yes!
- start with a recursive solution
- if inefficient, do you have **overlapping** subproblems?
- identify the **unique** subproblems
- Allocate an **array** to hold their solutions
  - K[] with size L, the knapsack capacity
- solve them from **bottom-up** smaller to larger
  - K[i] holds the maximum value possible with a knapsack of capacity *i*

Bottom-up solution

```
K[0] = 0
for (l = 1; l <= L; l++) {
     int max = 0;
     for (i = 0; i < n; i++) {
          if (wi <= l && vi + K[l - wi]) > max) {
               max = vi + K[l - wi];
          }
     }
     K[l] = max;
}
```

- **Runtime?**
  - **n * L**
    - **L's input size is in bits, hence:**
  - **$n * 2^{|L|}$**

Bottom-up Solution

weight:	6	3	4	2
value:	30	14	16	9

Size:	0	1	2	3	4	5	6	7	8	9	10
Max val:	0	0	9	14	18	23	30	32	39	44	48

Let’s summarize

- Greedy algorithms
  - elegant but hardly correct
  - need both optimal substructure and greedy choice
- Without the greedy choice property
  - have to solve all **unique** subproblems
  - can be done recursively using Memoization
  - or iteratively using dynamic programming

Where can we apply dynamic programming?

- Problems with two properties:
  - Optimal substructure
    - optimal solution contains optimal solutions of subproblems
  - Overlapping subproblems

Dynamic Programming: a recipe

- What is the **first decision** to make to solve the problem?
- What **subproblem**(s) emerge out of the that first decision?
- Must **wait** for subproblem solutions to make first decision
- start with a recursive solution
- if inefficient, do you have **overlapping** subproblems?
- identify the **unique** subproblems
- Allocate an **array** to hold their solutions
- solve them from **bottom-up** smaller to larger
- Optimize space if possible